A Workbench for Generation of Component Models

Marcus Blüml, Michael Lenzen, Adam Pawlak

Gesellschaft für Mathematik und Datenverarbeitung (GMD/SET)
Postfach 1316, 53731 St. Augustin, Germany

Abstract

We present a generator of behavioural component models which is highly flexible as it neither depends on a particular modelling technique, nor on a specific input format of a component specification. It is currently tuned for VHDL, but in fact is not HDL specific. To obtain a maximum degree of flexibility, the generator was designed as a model development environment basically composed of four module types: ***preprocessor modules*** *parsing and processing component specifications of a specific definition format,* ***method modules*** *representing the component modelling technique to be used, a* ***server module*** *that controls and invokes various generation activities, and finally a* ***client module*** *constituting the user interface. To provide a model developer with a workbench that can be tailored to his individual needs and grow with his experience is the major concern of our work.*

1. Introduction

Development of reliable models of hardware components becomes an element of production of electronic equipment. For those developing models for industry, standards for component model definition [Coe90, EIA, VITAL92] and computer support in a process of models' development have become an indispensable need. Tools, and especially model generators provide the only reasonable chance to apply appropriate modelling methods in such a huge engineering effort as development of model libraries is.

Literature on VHDL model generation does not provide many examples of operational systems. The best known are Gajski's SpecCharts and Modeller's Assistant [Sin90]. However these two are most predominantly concerned with a specification language used, SpecCharts and Process Model Graph respectively. Our goals are closer to those of the MODES modelling expert system [MODES91]. We have however concentrated on development of a workbench integrating different modelling rules including modelling conventions [EIA], which enables realization of our main goals, namely investigations on the compatibility of generated models, model generation techniques, and on the simulation efficiency of the generated VHDL code.

The workbench should support:

- An extendable set of component modelling techniques
- An extendable set of component specification formats
- Sharing of component specifications by different modelling techniques
- Generation of complete component models in contrast to partial models which have to be manually extended
- A powerful programming environment in order to ease maintenance

In this paper we describe the architecture of the workbench and its basic concepts, mainly the idea of an intermediate component representation and additional method specific information. We give an example to illustrate the information flow from a datasheet to the corresponding component model in VHDL.

2. Conceptual Background

Intermediate component specification language

Basically, a generator for component models is nothing but a compiler transforming instances *spec* of a source language *SPEC* into corresponding instances *hdl* of a target language *HDL*. Generators of this type are commonly used to support specific modelling techniques, seeing that component specifications usually depend on the technique being used. When different modelling techniques are used at the same time, the use of many generators and sets of component specifications for each of them is inadequate due to the enormous effort needed for this work. Therefore we use a uniform specification language as an intermediate format which is used by all generators. This means that each component needs to be specified only once but may be used to generate different component models corresponding to the different modelling techniques.

Modelling method specific information

It immediately turns out that, due to differences between modelling techniques, the use of a single specification language is not feasible in practice. Modelling techniques though conceptually similar in many cases, usually show differences that makes generation of all the details of the respective component models impossible. If one would try to develop a specification language that is powerful enough to handle all the information possibly needed, the complexity of hardware description languages would easily be reached. To avoid that, we decided to split all component specifications into two parts. One part is an instance of the so called *abstract component model (ACM)*, a specification language that was designed to specify basic information commonly needed by modelling methods. The other part, called *method specific information (MSI)*, is used to specify information that is not covered by the ACM but needed for model generation.

Preprocessors for different specification languages

Another important feature of our workbench is the concept of *preprocessor module (PM)* which allows a model developer to use different specification languages to create instances of the ACM. This concept is of relevance, as libraries of components may generaly be split into classes of component types with similar attributes. In order to simplify component specification, it is often feasible to define class-specific definition formats. Specifications written in those formats are transformed by respective preprocessor modules into a corresponding instance of the ACM. The existence of different definition formats is completely transparent for the modules which transform instances of the ACM into component models, as they only access the ACM but not the original specification. Figure 1 schematically shows the overall generation process.

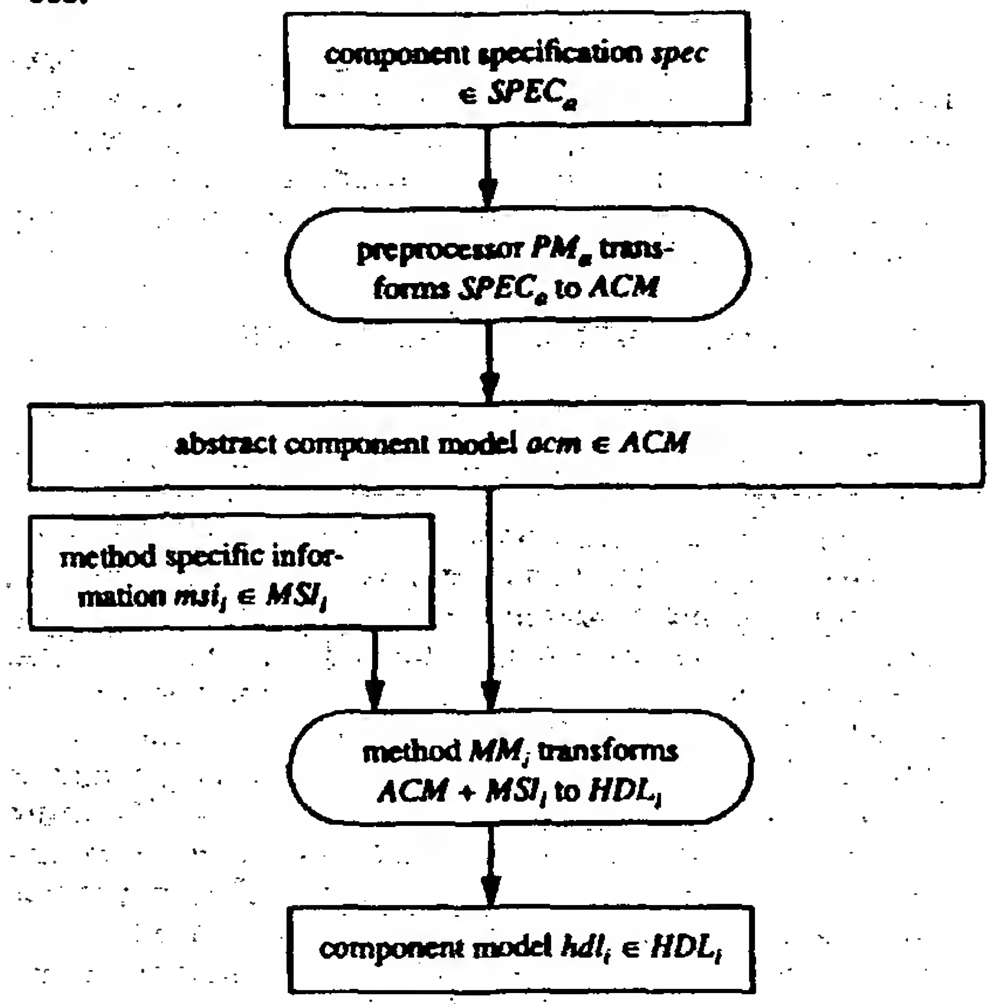

component specification *spec* $\in SPEC_a$
preprocessor PM_a transforms $SPEC_a$ to ACM
abstract component model *acm* $\in ACM$
method specific information $msi_j \in MSI_j$
method MM_j transforms $ACM + MSI_j$ to HDL_j
component model $hdl_j \in HDL_j$

Figure 1: Flow of Information from specification to component model

3. Abstract Component Model

This chapter describes the ACM more in detail, as this is the primary concept the whole system is based on. It must clearly stated that it is not directly available : it is an object that is accessed via a *functional interface* whenever a piece of information is requested. The example given in this article only represents instances of a particular specification language strongly related to the ACM.

ACM Concept

The main goal was to define an intermediate form for component representation with restricted functionality, which has both the expressiveness needed to model a comprehensive set of hardware components and the simplicity that allows its easy transformation in a HDL (not necessarily VHDL) representation according to a modelling technique. It is based on concepts which are found in common HDLs like VHDL, namely: concurrent processes with an internal state, shared signals used to communicate between processes, guarded commands that realize state changes and assertions that check correct component usage. Normally, most of the information needed to generate a component model is contained in the ACM, only minor additions are given via the MSI. Thus, it may be seen as the language which defines the set of components that is processable by our workbench.

Components and processes

We represent hardware as a set of concurrently operating indivisible units called processes that communicate via shared signals. We use a concept of module (called *component*) representing a subset of processes. This subset will be transformed by a so called *method module* to a component model using the module concept of the respective target HDL (e.g. the entity concept in VHDL). That is exactly what each instance of the ACM represents, namely an independent component consisting of an arbitrary number of processes. Each process performs a set of operations depending only on its current state and the current values of its input signals. We call these operations guarded commands, as their execution is triggered by boolean expressions.

Process transitions via guarded commands

It is assumed that hardware units perform only a limited set of operations. These operations are represented by an ordered collection of guarded commands that each process contains. The guard is the condition under which a corresponding set of operations is performed. Guarded commands are atomic in the sense that when they are performed all commands are executed without interruption by commands of other processes.

Limited set of commands and expressions

We allow only a limited set of commands that we assume to be powerful enough to model almost all components, in order to ease their transformation into a HDL. We restrict ourselves to the following features:

- conditional execution (if, then, else)
- fixed signal and variable types (boolean, bit, integer, real)
- delayed signal assignment
- check for signal change in expressions (only rising, falling)
- basic type conversion functions (e.g. integer value to list of bit values)

but particularly omit the following features, as their introduction would inhibit the use of a wider set of modelling techniques:

- repetitive execution (loops, while,...)
- user defined types (such as array, record)
- complex signal attributes (like stable in VHDL)
- VHDL code

We therefore have an intermediate specification format, that is not restricted to be transformed into VHDL, but may also be used with other HDLs even with conventional programming languages like C.

Signals, ports, value system

Basically, process communication is realized via signals that are shared between processes. In order to build modules of hardware that correspond to reusable component models, the concept of port is introduced. A port is nothing but a public signal with an assigned mode, either in, out or inout (lines 9 to 11 in § 5.2 Example 1). Each port may be used just like a private signal with the restrictions imposed by the mode. In order to support tristate and open collector logic, we introduced a value/strength system, associated with the type bit which allows for definition of signals that may be written by more than one process (like a resolved type in VHDL):

• F0,F1,FX	(forced low, high, undef)
• W0,W1,WX	(weak low, high, undef)
• Z	(high impedance)

The values of all sources (one for each writing processes) are mapped (like using a VHDL resolution function) to one of the following values: '0' (low), '1' (high), 'X' (undefined).

Delayed signal assignment, delay time declarations, operating conditions

In order to model timing behaviour, the ACM supports the use of a delayed signal assignment statement that references the declaration of a specific delay time. Delay time declarations allow sharing of identical delay times. They consist of a propagation delay and a load dependent delay part for each of the component's operating conditions (min, typ, max). They allow dividing of delay times within the ACM, and mapping to a single piece of source code within a generated component model. Operating conditions are used to support a concept which is found in many modelling techniques. The different operating conditions of a component do not need to be modelled explicitly, a single delayed signal assignment is sufficient.

Assertions

In addition to the component's pure functionality, restrictions on its use must usually be specified. The ACM allows for specification of following constraint types: setup violation, hold violation, check of pulse width and undefined signal. Again we use constraint time declarations to allow sharing of specific time values.

4. Architecture of the workbench

4.1 Implementation issues

We have chosen Prolog as the main implementation language for the workbench and Smalltalk to implement the user interface. Prolog's features backtracking and unification are very effectively usable in the traversal or transformation of complex data structures. Smalltalk is a highly interpretative (program changes allowed during runtime) object-oriented language. Usually implementations are complete development-environments with a comprehensive set of predefined classes. It allows the development and maintenance of complex interactive user interfaces with only little effort.

The generator implemented in **Quintus-Prolog** acts like a server, whereas the user interface implemented in **Objectworks-Smalltalk** (the client) manages the design information and completely controls the workbench. The system runs on a **Unix-workstation** and should be easily portable on any platform that runs the systems noted above.

4.2 Data model

We decided to leave management of objects which have to be stored permanently (e.g. component specifications) to a client application, as the generator itself should only provide basic services. Moreover, comfortable applications should be supplied by a user interface implemented in a language that is well suited to map interface concepts to concepts supplied by the generator. We introduced folders and categories to group component specifications as well as component models, both concepts that are unknown within the generator. Currently there are three user definable entities, shortly presented below.

Component specification

It represents the specification of a hardware component. It includes the name of a definition type that stands for a preprocessor module and the source code being transformed by the preprocessor module into the ACM. In fact, there should be exactly one such component specification for each datasheet of a component catalogue. Each component specification may be used by several component models.

Component model

Represents one specific component model. Any information that is used during source code generation of this component model is specified in this instance. It includes source code representing the MSI, a reference to the component specification and a reference to one of the configurations of the method module that is used to generate the HDL source code.

Method configuration

Method configurations represent variations of modelling techniques. Instead of defining one specific method module for each of these variations we specify those options as a piece of source code. Method configurations are owned by a method, as they are processed by the respective method module, syntax may therefore depend on the method they are owned by.

4.3 Modules

Each module represents a collection of Prolog source files, they are grouped by functionality. In case of multiple instances (e.g. preprocessor modules) only one at a time is active.

Client module

Modules of this type are no element of the generator itself, they instead represent applications that access the generator resources. An example is a network-module that offers functionalities to the server via a communication mecha-

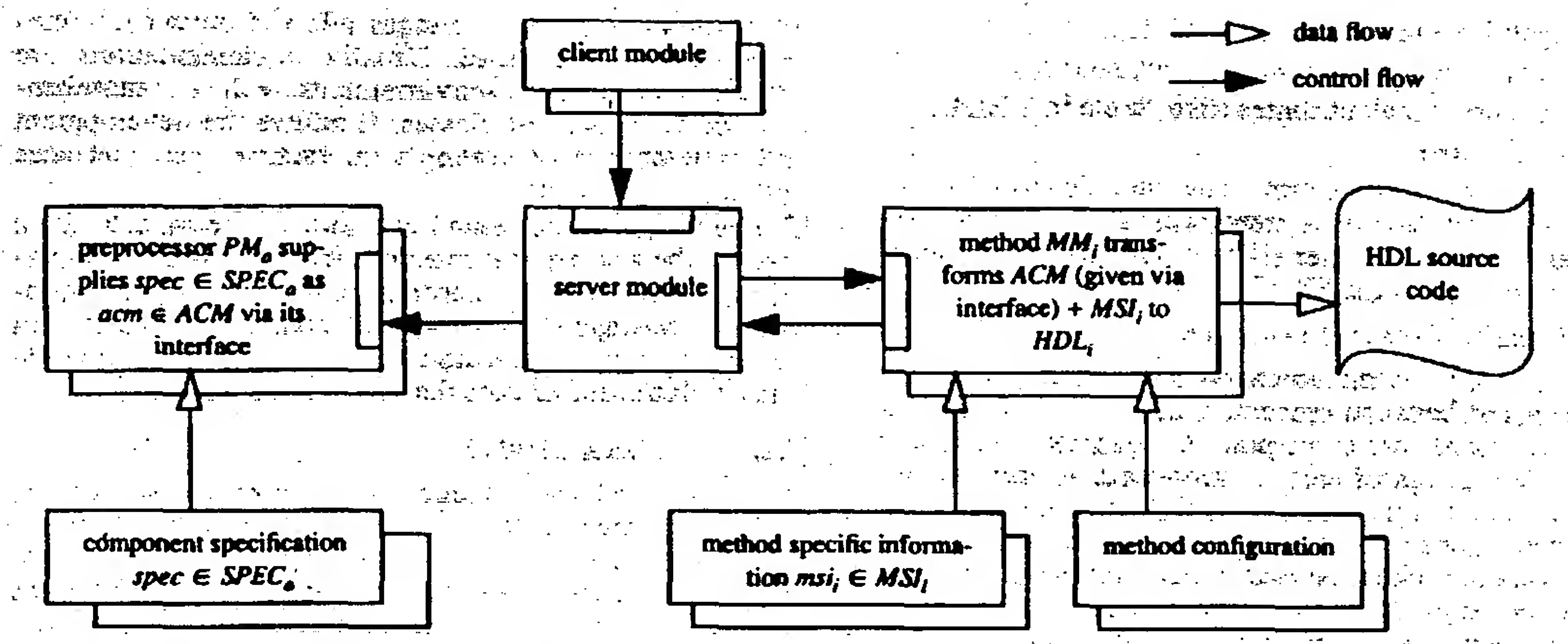

client module
data flow
control flow
preprocessor PM_a supplies $spec \in SPEC_a$ as $acm \in ACM$ via its interface
server module
method MM_i transforms ACM (given via interface) + MSI_i to HDL_i
HDL source code
component specification $spec \in SPEC_a$
method specific information $msi_i \in MSI_i$
method configuration

Figure 2: Control and dataflow in the workbench

nism of the operating system. In our case this interface is used by a browser implemented in Objectworks-Smalltalk.

Server module

This module coordinates the invocation of functions within the other modules and guarantees the predefined protocol. It furthermore maps higher level function calls into lower level calls and simplifies the implementation of services within the preprocessor modules.

Preprocessor modules

Those modules represent the specification types that are referenced by component specifications. Before a component model could be generated, the preprocessor module of the referenced component specification has to be consulted so that the component specification can be processed by that module. Each preprocessor module has to supply a functional interface (a set of predicates) that is used to access the ACM.

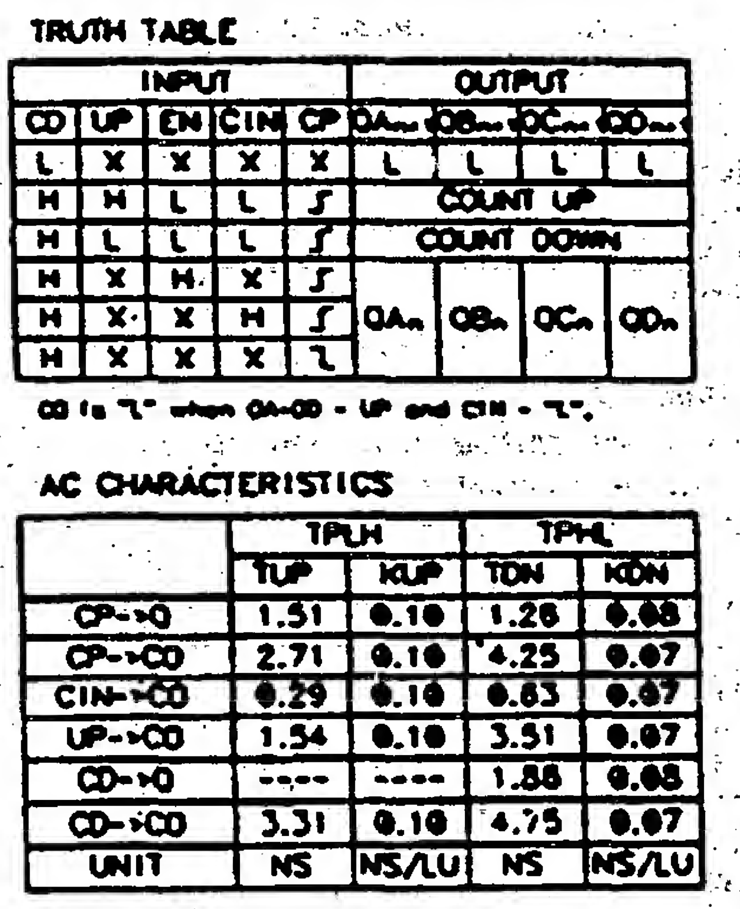
Method modules

These modules represent modelling techniques. Before a component model referencing a specific method configuration could be generated, the corresponding method module has to be consulted. Method modules transform a component specification given as the ACM and MSI into the corresponding model in a HDL.

5. Example

5.1 Datasheet of Standard Cell Library Item [Sie89]

Figure 3 constitutes a part of a datasheet of a standard cell catalogue. It illustrates the way functionality and timing are given at this abstraction level. This datasheet corresponds to the component specification given in § 5.2



TRUTH TABLE

INPUT					OUTPUT			
CD	UP	EN	CIN	CP	QA_{n+1}	QB_{n+1}	QC_{n+1}	QD_{n+1}
L	X	X	X	X	L	L	L	L
H	H	L	L	↑	COUNT UP			
H	L	L	L	↑	COUNT DOWN			
H	X	H	X	↑	QA_n	QB_n	QC_n	QD_n
H	X	X	H	↑	QA_n	QB_n	QC_n	QD_n
H	X	X	X	↓	QA_n	QB_n	QC_n	QD_n

CO is "L" when QA-QD = UP and CIN = "L".

AC CHARACTERISTICS

	TPLH		TPHL	
	TUP	KUP	TDN	KDN
CP→Q	1.51	0.10	1.26	0.08
CP→CO	2.71	0.10	4.25	0.07
CIN→CO	0.29	0.10	0.63	0.07
UP→CO	1.54	0.10	3.51	0.07
CD→Q	-----	-----	1.86	0.08
CD→CO	3.31	0.10	4.75	0.07
UNIT	NS	NS/LU	NS	NS/LU

Figure 3: Datasheet of 4-bit counter

5.2 Component specification

The following code shows the specification of a counter. This specification format was specially designed to mirror directly the concepts of the ACM. Prolog-like syntax is used in order to simplify parsing (done by the corresponding preprocessor module).

Lines 8 to 30 define a process; three guarded commands (beginning with 'when') are shown. The guarded command in line 22 to 29 is transformed to the VHDL source-code shown in Example 3 from line 25 to 41. In line 23 (Example 1) a macro, defined in lines 37 to 39, is called (macros are a feature of the preprocessor, not of the ACM). This illustrates one possibility of metaprogramming which may easily be used with Prolog.

```
component 'CUD42'
    name :- 'CUD42'.
    comment :- '4-bit UP/DOWN-COUNTER with
            CLEAR, EXPANDABLE'.
    port(cd) :- modus in, type ctl, name 'CD',
            comment 'clear'.
    port(qa) :- modus out, type bit, name 'QA',
            comment 'counter bit 1'.
    /* other ports analogous */
begin.
process beh.
    port(cd) :- modus in.
    port(qa) :- modus out.
    /* analogous for other process ports */
    var(vq) :- type integer, init 0.
    var(vco) :- type bit, init '0'.
    /* same for vqa..vqd */
begin.
when(cd='0') :-
    vq := 0,
    /* analogous to guarded command below*/
when((cd='1' and up='1' and en='0' and cin='0'
            and rise(cp))) :-
    call succ(vq,16),
    /* analogous to guarded command below*/
when((cd='1' and up='0' and en='0' and cin='0'
            and rise(cp))) :-
    call pred(vq,16),
    vco := bit(not(vqa=up and vqb=up and vqc=up
            and vqd=up and cin='0')),
    intToBit(vq,[vqa,vqb,vqc,vqd]),
    if(vco='0',
        assign(co, force(vco),
            delay((cd,co,hl)),
        assign(co, force(vco),
            delay((cd,co,lh))),
    /* analogous for qa..qd */
end beh.

operate(typ) :- name 'TYP', comment 'typical
            operating condition'.

delay((cp,q,lh)) :- name 'CP_Q_LH',
            prop(typ,1510), load(typ,100).
constraint((up,cp)):- type setup, name
            'SE_UP', setup(typ,4960).
        ...
macro(pred(INT,MODULO)) :-
    MOD1 is MODULO-1,
    INT := if(INT>0,INT-1,MOD1).
        ...
assert((up,cp)) :- setup(up,cp),
    when(cd='1' and en='0' and rise(cp)),
    name 'UP_CP', constraint((up,cp)).
        ...
end 'CUD42'.
```

Example 1: Component specification of 4-bit counter

```
operate(typ,typ). /*All possible types mapped*/
operate(min,typ). /*to the type typ*/
operate(max,typ).
act(cd,low).  /*Activities required by the*/
act(up,data). /*timing checks*/
act(en,low).
act(cin,low).
act(cp,rising).
```

Example 2: MSI data for 4-bit counter required by GMD modelling technique [BKLP92]

5.3 VHDL source code

```
ENTITY CUD42_g IS -- 4-bit UP/DOWN-COUNTER
            with CLEAR, EXPANDABLE
GENERIC(...);
PORT(
   g_CD:IN    cl_wlogic;  -- clear
   g_UP:IN    cl_wlogic;  -- up/down selec
   g_QA:OUT   cl_wlogic;  -- counter bit 1
   ...);
...
END CUD42_g;

ARCHITECTURE gen OF CUD42_g IS
FU_beh: PROCESS(g_CD,g_UP,g_EN,g_CIN,g_CP)
   VARIABLE v_vq : INTEGER:=0;
   VARIABLE v_vco : cl_logic:=cl_0;
   ...
   VARIABLE inter : cl_logic_v(0 to 31);
BEGIN
   no_id <= FALSE;
   IF (g_CD = cl_0) THEN
      v_vq := 0;
      ...
   ELSIF ((g_CD = cl_1) AND ((g_UP = cl_1) AND
         ((g_EN = cl_0) AND ((g_CIN = cl_0)
         AND cl_rising(g_CP))))) THEN
      v_vq := cl_if_int(((v_vq + 1) <
         16),(v_vq + 1),0);

   ELSIF ((g_CD = cl_1) AND ((g_UP = cl_0) AND
         ((g_EN = cl_0) AND ((g_CIN = cl_0)
         AND cl_rising(g_CP))))) THEN
      v_vq := cl_if_int((v_vq > 0),(v_vq -
         1),15);
      v_vco := cl_bit((NOT ((v_vqa = g_UP) AND
         ((v_vqb = g_UP) AND ((v_vqc = g_UP)
         AND ((v_vqd = g_UP) AND (g_CIN =
         cl_0)))))));
      cl_int2lo(
            d => v_vq,
            s => inter);
      v_vqa:=inter(31);
      v_vqb:=inter(30);
      v_vqc:=inter(29);
      v_vqd:=inter(28);
      IF (v_vco = cl_0) THEN
            g_CO <= cl_force(0);
            del_CO<=4.25E+00 ns;
      ELSE
            g_CO <= cl_force(1);
            del_CO<=2.71E+00 ns;
      END IF;
   ...
   END IF;
END PROCESS;

END gen;

```

Example 3: Generated VHDL code

6. Performance data

In order to get first aproximation of the system's performance we experimented with standard cell models of:

• AN2	2-input AND Gate
• DFFRS	D-flip-flop with Preset & Set
• CUD42	4-bit Up/Down-counter with clear, expandable

The modelling technique developed at Thomson-CSF [Thom92] requires more method specific information as it allows for electrical and physical characteristics (voltage, current, temperature, protection class,...) as well as for pin numbers. Nevertheless, method specific information is in any case very limited in comparison to the common component specification that is to be transformed into the ACM, as it is visible from Examples 1 and 2.

Table 1: Static resources

Module	kbyte	lines of code
Method module for modelling technique developed at GMD	51.1	1600
Method module for modelling technique developed at Thomsom CSF	66.6	1800
Preprocessor module	28.5	1000
Server module	13.0	550
User interface-Smalltalk	75.8	2000

Table 2: Dynamic resources

Component	AN2	DFFRS	CUD42
Component specification (kbyte)	1.1	3.7	4.9
Modelling technique developed at GMD			
Method specific information (kbyte)	0.05	0.12	0.12
Generated VHDL source code (kbyte)	7.5	22.6	34.9
Total CPU time used for generation (sec)	3.2	6.5	10.4
Modelling technique developed at Thomsom-CSF			
Method specific information (kbyte)	0.22	0.31	0.30
Generated VHDL source code (kbyte)	11.6	27.4	34.9
Total CPU time used for generation (sec)	4.5	8.1	10.0

7. Summary

A workbench for development of component models has been presented. It provides for:

- generation of complete behavioural component models
- integration of different component modelling techniques (deep knowledge of both the workbench and the modelling technique is required)
- sharing of specifications using an extendable set of component definition formats (knowledge of the internal component representation is required)

By splitting specifications into two parts, we were able to generate complete component models using single specifications (the ACM) augmented by information (the MSI) specific for each component modelling technique used.

The programming environment which was used (Quintus-Prolog and Objectworks-Smalltalk) has proven to be adequate for our purposes, particularly robustness, programming efficiency and runtime performance were highly satisfactory.

We are going to extend the system in the following respects:

- Available preprocessors are tuned for specification of standard cells. Modelling rules for complex components have to be studied, especially in relation to their simulation efficiency; respective method and preprocessor modules are to be defined
- An extension of the client module interface would allow stronger coupling of client and server and should make possible an improved integration of user interface and generator

Acknowledgements

We would like to thank our partners from ECIP II WP2 for fruitful discussions on component modelling techniques. In particular, we acknowledge the permission of Thomson-CSF to apply their modelling methodology. Finally, we thank Frédérique Bouchard from IRESTE-Nantes for writing the method module based on Thomson's rules and profoundly experimenting with the generator.

References

[MODES91] H. P. Amann et al., *MODES: An Expert System for the Automatic Generation of Behavioural Hardware Models*, Proc. of EURO-VHDL'91, Stockholm.

[BKLP92] M. Blüml, et al., *A Methodology for the Development of High Quality Standard-Cell Models in VHDL*, VHDL Forum, Spring'92, Santander, April 1992.

[BBP] M.Blüml, F.Bouchard, A.Pawlak, *A Technique For a Flexible Generation of Component Models*, paper in preparation.

[Coe90] D. Coelho, *Follow Simple Rules to Create VHDL Models*, Electronic Design, June 14, 1990.

[EIA] Electronic Industries Association, *Commercial Component Model Specification*, Revision J, May 1989 and June 1992.

[FLP92] T. Feltin, M. Lenzen, A. Pawlak, *Technological Aspects in VHDL Models of Standard Components*, PATMOS Workshop, Paris, Sept. 1992.

[Sie89] Siemens A.G., *SCoD ADVANCLL D - Standard Cells*, 1989.

[Sin90] B. Singh, *A Parametrized CAD Tool for VHDL Model Development with X Windows*, Report Virginia Tech.

[Thom92] Thomson-CSF, *VHDL For Component Modelling*, ESPRIT Project 2072 ECIP Report, Dec. 1992.

[VITAL92] VITAL: *VHDL Initiative Toward ASIC Libraries*, Technical Requirements, Version 1.0, Oct. 1992.